C++ tutorial for C users

This text shows and highlights features and basic principles of C++. It is aimed at experienced C users who wish to learn C++. You will be able to express your code using the richer C++ syntax and you will be able to read some C++ code.

Though the concept is introduced and made usable, this is not a tutorial about Object Oriented Programming. You will still have to learn the spirit of OOP and the details of its implementation in C++, in order to be a true C++ programmer.

1. A new way to include libraries
2. // for one-line remarks
3. Console input and output streams
4. Variable declarations can be put inside the code without using hooks
5. Variables can be initialised by a calculation involving other variables
6. Variables can be declared inside a **for** loop declaration
7. Global variables can be accessed even if a local variables have the same name
8. It is possible to declare a REFERENCE to another variable
9. Namespaces can be declared
10. A function can be declared **inline**
11. The **exception** structure has been added
12. A function can have default parameters
13. FUNCTION OVERLOAD: several functions can be declared with the same name provided there is a difference in their parameter list
14. The symbolic operators (+ - * / ...) can be defined for new data types
15. Different functions for different data types will automatically be generated provided you define a **template** function
16. The keywords **new** and **delete** are much better to allocate and deallocate memory
17. You can add METHODS To a **class** or struct

18. The CONSTRUCTOR and the DESTRUCTOR can be used to initialise and destroy an instance of a class
19. Complex classes need the COPY CONSTRUCTOR and an overload of the = operator
20. The method bodies can be defined below the class definition (and Makefile usage example)
21. The keyword **this** is a pointer to the instance a method is acting upon
22. Arrays of instances can be declared
23. An example of a complete class declaration
24. **static** variables inside a class definition
25. **const** variables inside a class definition
26. A class can be DERIVED from another class
27. If a method is declared **virtual** the program will always check the type of the instance that is pointed to and will use the appropriate method.
28. A class can be derived from more than one base class
29. Class derivation allows you to write generic methods
30. ENCAPSULATION: **public**, **protected** and **private**
31. Brief examples of file I/O
32. Character arrays can be used like files
33. An example of formatted output

1.

There is a new way to **#include** libraries (the old method still works yet the compiler complains). The **.h** extension is not used any more, and the names of standard C libraries are written beginning with a **c**. In order for the program to use these libraries correctly **using namespace std;** has to be added:

```
using namespace std;
#include <iostream>         // This is a key C++ library
#include <cmath>            // The standard C library math.h

int main ()
{
   double a;
```

```
   a = 1.2;
   a = sin (a);

   cout << a << endl;

   return 0;
}
```

Output
0.932039

A few hints for beginners:

To compile this program, type it (or copy & paste it) into a text editor (gedit, kwrite, kate, kedit, vi, emacs, nano, pico, mcedit, Notepad...), save it as a file named, say test01.cpp (if you are a newbie, best put this file inside your home directory, that is, for example /home/jones on a Unix-like box).

To compile the source code file, type this command (on most open-source Unix-like boxes) in a console or terminal window:

```
g++ test01.cpp -o test01
```

To run the binary executable file test01 that has been produced by the compilation (assuming there were no errors), type this:

```
./test01
```

Each time you modify the test01.cpp source code file, you need to compile it again if you want the modifications to be reflected in the test01 executable file (type the up-arrow key on your keyboard to recall commands).

2.

You can use // to type a remark:

```
using namespace std;          // Using the standard library namespace.
#include <iostream>           // The iostream library is often used.

int main ()                   // The program's main routine.
{
   double a;                  // Declaration of variable a.

   a = 456.47;
   a = a + a * 21.5 / 100;    // A calculation.

   cout << a << endl;         // Display the content of a.

   return 0;                  // Program end.
```

```
}
```

Output
554.611

(The possibility to use // to type remarks has been added to C in C99 and ANSI C 2000.)

3.

Input from the keyboard and output to the screen can be performed through **cout** << and **cin** >>:

```
using namespace std;
#include <iostream>

int main()
{
   int a;                    // a is an integer variable
   char s [100];             // s points to a string of max 99 characters

   cout << "This is a sample program." << endl;

   cout << endl;             // Just a line feed (end of line)

   cout << "Type your age : ";
   cin >> a;

   cout << "Type your name: ";
   cin >> s;

   cout << endl;

   cout << "Hello " << s << " you're " << a << " old." << endl;
   cout << endl << endl << "Bye!" << endl;

   return 0;
}
```

Output
This is a sample program. Type your age : 12 Type your name: Edmond Hello Edmond you're 12 old.   Bye!

4.

Variables can be declared anywhere inside the code:

```
using namespace std;
#include <iostream>

int main ()
{
   double a;

   cout << "Hello, this is a test program." << endl;

   cout << "Type parameter a: ";
   cin >> a;

   a = (a + 1) / 2;

   double c;

   c = a * 5 + 1;

   cout << "c contains      : " << c << endl;

   int i, j;

   i = 0;
   j = i + 1;

   cout << "j contains      : " << j << endl;

   return 0;
}
```

Output
Hello, this is a test program. Type parameter a: 7 c contains : 21 j contains : 1

Maybe try to use this feature to make your source code more readable and not to mess it up.
Like in C, variables can be encapsulated between { } blocks. Then they are local in scope to the zone encapsulated between the { and }. Whatever happens with such variables inside the encapsulated zone will have no effect outside the zone:

```
using namespace std;
#include <iostream>
```

```
int main ()
{
   double a;

   cout << "Type a number: ";
   cin >> a;

   {
      int a = 1;
      a = a * 10 + 4;
      cout << "Local number: " << a << endl;
   }

   cout << "You typed: " << a << endl;

   return 0;
}
```

Output

```
Type a number: 9
Local number: 14
You typed: 9
```

5.

A variable can be initialised by a calculation involving other variables:

```
using namespace std;
#include <iostream>

int main ()
{
   double a = 12 * 3.25;
   double b = a + 1.112;

   cout << "a contains: " << a << endl;
   cout << "b contains: " << b << endl;

   a = a * 2 + b;

   double c = a + b * a;

   cout << "c contains: " << c << endl;

   return 0;
}
```

Output

```
a contains: 39
b contains: 40.112
c contains: 4855.82
```

6.

C++ allows you to declare a variable to be local to a loop:

```
using namespace std;
#include <iostream>

int main ()
{
   int i;                        // Simple declaration of i
   i = 487;

   for (int i = 0; i < 4; i++)   // Local declaration of i
   {
      cout << i << endl;         // This outputs 0, 1, 2 and 3
   }

   cout << i << endl;            // This outputs 487

   return 0;
}
```

Output

```
0
1
2
3
487
```

In case the variable is not declared somewhere above the loop, you may be tempted to use it below the loop. Some early C++ compilers accept this. Then the variable has the value it had when the loop ended. You shouldn't do this. It's considered bad practice:

```
using namespace std;
#include <iostream>

int main ()
{

   for (int i = 0; i < 4; i++)
   {
      cout << i << endl;
   }
```

```
   cout << i << endl;           // Bad practice!
   i += 5;                      // Bad practice!
   cout << i << endl;           // Bad practice!

   return 0;
}
```

Gnu C++ compiler complain
t.cpp: In function ‘int main()’: t.cpp:12: error: name lookup of ‘i’ changed for new ISO ‘for’ scoping t.cpp:7: error: using obsolete binding at ‘i’

7.

A global variable can be accessed even if another variable with the same name has been declared inside the function:

```
using namespace std;
#include <iostream>

double a = 128;

int main ()
{
   double a = 256;

   cout << "Local a:  " << a   << endl;
   cout << "Global a: " << ::a << endl;

   return 0;
}
```

Output
Local a: 256 Global a: 128

8.

It is possible to make one variable be another:

```
using namespace std;
#include <iostream>

int main ()
{
   double a = 3.1415927;

   double &b = a;                              // b is a

   b = 89;

   cout << "a contains: " << a << endl;     // Displays 89.

   return 0;
}
```

Output
a contains: 89

(If you are used to pointers and absolutely want to know what happens, simply think **double &b = a** is translated to **double *b = &a** and all subsequent **b** are replaced by ***b**.)

The value of REFERENCE **b** cannot be changed after its declaration. For example you cannot write, a few lines further, **&b = c** expecting that **b** is now **c**. It won't work. Everything is said on the declaration line of **b**. Reference **b** and variable **a** are married on that line and nothing will separate them.

References can be used to allow a function to modify a calling variable:

```
using namespace std;
#include <iostream>

void change (double &r, double s)
{
   r = 100;
   s = 200;
}

int main ()
{
   double k, m;

   k = 3;
   m = 4;

   change (k, m);

   cout << k << ", " << m << endl;         // Displays 100, 4.

   return 0;
}
```

Output
100, 4

If you are used to pointers in C and wonder how exactly the program above works, here is how the C++ compiler would translate it to C:

```
using namespace std;
#include <iostream>

void change (double *r, double s)
{
   *r = 100;
   s = 200;
}

int main ()
{
   double k, m;

   k = 3;
   m = 4;

   change (&k, m);

   cout << k << ", " << m << endl;        // Displays 100, 4.

   return 0;
}
```

Output
100, 4

A reference can be used to let a function return a variable:

```
using namespace std;
#include <iostream>

double &biggest (double &r, double &s)
{
   if (r > s) return r;
   else       return s;
}

int main ()
{
   double k = 3;
   double m = 7;

   cout << "k: " << k << endl;       // Displays  3
   cout << "m: " << m << endl;       // Displays  7
   cout << endl;

   biggest (k, m) = 10;

   cout << "k: " << k << endl;       // Displays  3
   cout << "m: " << m << endl;       // Displays 10
   cout << endl;
```

```
   biggest (k, m) ++;

   cout << "k: " << k << endl;       // Displays  3
   cout << "m: " << m << endl;       // Displays 11
   cout << endl;

   return 0;
}
```

Output

```
k: 3
m: 7

k: 3
m: 10

k: 3
m: 11
```

Again, provided you're used to pointer arithmetic and if you wonder how the program above works, just imagine that the compiler translated it into the following standard C program:

```
using namespace std;
#include <iostream>

double *biggest (double *r, double *s)
{
   if (*r > *s) return r;
   else         return s;
}

int main ()
{
   double k = 3;
   double m = 7;

   cout << "k: " << k << endl;
   cout << "m: " << m << endl;
   cout << endl;

   (*(biggest (&k, &m))) = 10;

   cout << "k: " << k << endl;
   cout << "m: " << m << endl;
   cout << endl;

   (*(biggest (&k, &m))) ++;

   cout << "k: " << k << endl;
   cout << "m: " << m << endl;
   cout << endl;

   return 0;
}
```

Output

```
k: 3
m: 7

k: 3
m: 10

k: 3
m: 11
```

To end with, for people who have to deal with pointers yet do not like it, references are useful to effectively un-pointer variables. Beware this is considered bad practice. You can get into trouble. See for example http://www.embedded.com/story/OEG20010311S0024.

```
using namespace std;
#include <iostream>

double *silly_function ()    // This function returns a pointer to a double
{
   static double r = 342;
   return &r;
}

int main ()
{
   double *a;

   a = silly_function();

   double &b = *a;          // Now b is the double towards which a points!

   b += 1;                  // Great!
   b = b * b;               // No need to write *a everywhere!
   b += 4;

   cout << "Content of *a, b and r: " << b << endl;

   return 0;
}
```

Output

```
Content of *a, b and r: 117653
```

9.

Namespaces can be declared. The variables declared within a namespace can be used thanks to the **::** operator:

```
using namespace std;
#include <iostream>
#include <cmath>

namespace first
{
   int a;
   int b;
}

namespace second
{
   double a;
   double b;
}

int main ()
{
   first::a = 2;
   first::b = 5;

   second::a = 6.453;
   second::b = 4.1e4;

   cout << first::a + second::a << endl;
   cout << first::b + second::b << endl;

   return 0;
}
```

Output
8.453 41005

10.

If a function contains just simple lines of code, doesn't use **for** loops or the like, it can be declared **inline**. This means its code will be inserted everywhere the function is used. That's somewhat like a macro. The main advantage is the program will be faster. A small drawback is it will be bigger, because the full code of the function was inserted everywhere it is used:

```
using namespace std;
#include <iostream>
#include <cmath>

inline double hypothenuse (double a, double b)
{
   return sqrt (a * a + b * b);
}

int main ()
```

```
{
   double k = 6, m = 9;

   // Next two lines produce exactly the same code:

   cout << hypothenuse (k, m) << endl;
   cout << sqrt (k * k + m * m) << endl;

   return 0;
}
```

Output
10.8167 10.8167

(Inline functions have been added to C in C99 and ANSI C 2000.)

11.

You know the classical control structures of C: **for**, **if**, **do**, **while**, **switch**... C++ adds one more control structure named EXCEPTION:

```
using namespace std;
#include <iostream>
#include <cmath>

int main ()
{
   int a, b;

   cout << "Type a number: ";
   cin >> a;
   cout << endl;

   try
   {
      if (a > 100) throw 100;
      if (a < 10)  throw 10;
      throw a / 3;
   }
   catch (int result)
   {
      cout << "Result is: " << result << endl;
      b = result + 1;
   }

   cout << "b contains: " << b << endl;

   cout << endl;
```

```
   // another example of exception use:

   char zero []     = "zero";
   char pair []     = "pair";
   char notprime [] = "not prime";
   char prime []    = "prime";

   try
   {
      if (a == 0) throw zero;
      if ((a / 2) * 2 == a) throw pair;
      for (int i = 3; i <= sqrt (a); i++)
      {
         if ((a / i) * i == a) throw notprime;
      }
      throw prime;
   }
   catch (char *conclusion)
   {
      cout << "The number you typed is "<< conclusion << endl;
   }

   cout << endl;

   return 0;
}
```

Output

```
Type a number: 5

Result is: 10
b contains: 11

The number you typed is prime
```

12.

It is possible to define default parameters for functions:

```
using namespace std;
#include <iostream>

double test (double a, double b = 7)
{
   return a - b;
}

int main ()
{
   cout << test (14, 5) << endl;     // Displays 14 - 5
   cout << test (14) << endl;        // Displays 14 - 7
```

```
   return 0;
}
```

Output
9 7

13.

One important advantage of C++ is the FUNCTION OVERLOAD. Different functions can have the same name provided something allows the compiler to distinguish between them: number of parameters, type of parameters...

```
using namespace std;
#include <iostream>

double test (double a, double b)
{
   return a + b;
}

int test (int a, int b)
{
   return a - b;
}

int main ()
{
   double   m = 7,  n = 4;
   int      k = 5,  p = 3;

   cout << test(m, n) << " , " << test(k, p) << endl;

   return 0;
}
```

Output
11 , 2

14.

OPERATOR OVERLOADING can be used to redefine the basic symbolic operators for new kinds of parameters:

```
using namespace std;
#include <iostream>

struct vector
{
   double x;
   double y;
};

vector operator * (double a, vector b)
{
   vector r;

   r.x = a * b.x;
   r.y = a * b.y;

   return r;
}

int main ()
{
   vector k, m;               // No need to type "struct vector"

   k.x =  2;                  // To be able to write
   k.y = -1;                  // k = vector (2, -1)
                              // see chapter 19.

   m = 3.1415927 * k;         // Magic!

   cout << "(" << m.x << ", " << m.y << ")" << endl;

   return 0;
}
```

Output
(6.28319, -3.14159)

Besides multiplication, 43 other basic C++ operators can be overloaded, including +=, ++, the array **[]**, and so on...

The << operator, normally used for binary shifting of integers, can be overloaded to give output to a stream instead (e.g., **cout** <<). It is possible to overload the << operator further for the output of new data types, like vectors:

```
using namespace std;
#include <iostream>

struct vector
{
   double x;
   double y;
};

ostream& operator << (ostream& o, vector a)
{
   o << "(" << a.x << ", " << a.y << ")";
   return o;
```

```
}

int main ()
{
   vector a;

   a.x = 35;
   a.y = 23;

   cout << a << endl;      // Displays (35, 23)

   return 0;
}
```

Output
(35, 23)

15.

Tired of defining the same function five times? One definition for **int** type parameters, one definition for **double** type parameters, one definition for **float** type parameters... Didn't you forget one type? What if a new data type is used? No problem: the C++ compiler can automatically generate every version of the function that is necessary! Just tell it how the function looks like by declaring a **template** function:

```
using namespace std;
#include <iostream>

template <class ttype>
ttype minimum (ttype a, ttype b)
{
   ttype r;

   r = a;
   if (b < a) r = b;

   return r;
}

int main ()
{
   int i1, i2, i3;
   i1 = 34;
   i2 = 6;
   i3 = minimum (i1, i2);
   cout << "Most little: " << i3 << endl;

   double d1, d2, d3;
   d1 = 7.9;
   d2 = 32.1;
   d3 = minimum (d1, d2);
   cout << "Most little: " << d3 << endl;

   cout << "Most little: " << minimum (d3, 3.5) << endl;
```

```
   return 0;
}
```

Output
Most little: 6 Most little: 7.9 Most little: 3.5

The function **minimum** is used three times in the above program, yet the C++ compiler generates only two versions of it: **int minimum (int a, int b)** and **double minimum (double a, double b)**. That does the job for the whole program.

What would happen if you tried something like calculating **minimum (i1, d1)**? The compiler would have reported that as an error. That's because the template states that both parameters are of the same type.

You can use an arbitrary number of different template data types in a template definition. And not all the parameter types must be templates, some of them can be of standard types or user defined (**char**, **int**, **double**...). Here is an example where the **minimum** function takes parameters of any type (different or the same) and outputs a value that has the type of the first parameter:

```
using namespace std;
#include <iostream>

template <class type1, class type2>
type1 minimum (type1 a, type2 b)
{
   type1 r, b_converted;
   r = a;
   b_converted = (type1) b;
   if (b_converted < a) r = b_converted;
   return r;
}

int main ()
{
   int i;
   double d;

   i = 45;
   d = 7.41;

   cout << "Most little: " << minimum (i, d) << endl;
   cout << "Most little: " << minimum (d, i) << endl;
   cout << "Most little: " << minimum ('A', i) << endl;

   return 0;
}
```

Output
Most little: 7 Most little: 7.41 Most little: -

(The ASCII code of character '-' is 45 while the code of 'A' is 65.)

16.

The keywords **new** and **delete** can be used to allocate and deallocate memory. They are cleaner than the functions **malloc** and **free** from standard C.

new [] and **delete []** are used for arrays.

```
using namespace std;
#include <iostream>
#include <cstring>

int main ()
{
   double *d;                          // d is a variable whose purpose
                                       // is to contain the address of a
                                       // zone where a double is located


   d = new double;                     // new allocates a zone of memory
                                       // large enough to contain a double
                                       // and returns its address.
                                       // That address is stored in d.

   *d = 45.3;                          // The number 45.3 is stored
                                       // inside the memory zone
                                       // whose address is given by d.

   cout << "Type a number: ";
   cin >> *d;

   *d = *d + 5;

   cout << "Result: " << *d << endl;

   delete d;                           // delete deallocates the
                                       // zone of memory whose address
                                       // is given by pointer d.
                                       // Now we can no more use that zone.


   d = new double[15];                 // allocates a zone for an array
                                       // of 15 doubles. Note each 15
                                       // double will be constructed.
                                       // This is pointless here but it
                                       // is vital when using a data type
                                       // that needs its constructor be
                                       // used for each instance.

   d[0] = 4456;
   d[1] = d[0] + 567;

   cout << "Content of d[1]: " << d[1] << endl;

   delete [] d;                        // delete [] will deallocate the
```

```
                                    // memory zone. Note each 15
                                    // double will be destructed.
                                    // This is pointless here but it
                                    // is vital when using a data type
                                    // that needs its destructor be
                                    // used for each instance (the ~
                                    // method). Using delete without
                                    // the [] would deallocate the
                                    // memory zone without destructing
                                    // each of the 15 instances. That
                                    // would cause memory leakage.

   int n = 30;

   d = new double[n];               // new can be used to allocate an
                                    // array of random size.
   for (int i = 0; i < n; i++)
   {
      d[i] = i;
   }

   delete [] d;


   char *s;

   s = new char[100];

   strcpy (s, "Hello!");

   cout << s << endl;

   delete [] s;

   return 0;
}
```

Output
Type a number: 6 Result: 11 Content of d[1]: 5023 Hello!

17.

In standard C a **struct** contains only data. In C++ a struct definition can also include functions. Those functions are owned by the struct and are meant to operate on the data of the struct. Those functions are called METHODS. The example below defines the method **surface()** on the struct **vector**:

```
using namespace std;
#include <iostream>

struct vector
```

```
{
   double x;
   double y;

   double surface ()
   {
      double s;
      s = x * y;
      if (s < 0) s = -s;
      return s;
   }
};

int main ()
{
   vector a;

   a.x = 3;
   a.y = 4;

   cout << "The surface of a: " << a.surface() << endl;

   return 0;
}
```

Output
The surface of a: 12

In the example above, **a** is an INSTANCE of struct "vector". (Note that the keyword "**struct**" was not necessary when declaring vector **a**.)

Just like a function, a method can be an overload of any C++ operator, have any number of parameters (yet one parameter is always implicit: the instance it acts upon), return any type of parameter, or return no parameter at all.

What is a **class**? It's a **struct** that keeps its data hidden. Only the methods of the **class** can access the data. You can't access the data directly, unless authorized by the **public:** directive. Here is an example of a **class** definition. It behaves exactly the same way as the **struct** example above because the class data **x** and **y** are defined as public:

```
using namespace std;
#include <iostream>

class vector
{
public:

   double x;
   double y;

   double surface ()
   {
      double s;
      s = x * y;
      if (s < 0) s = -s;
      return s;
   }
};
```

```
int main ()
{
   vector a;

   a.x = 3;
   a.y = 4;

   cout << "The surface of a: " << a.surface() << endl;

   return 0;
}
```

Output
The surface of a: 12

In the example above, the **main()** function changes the data of instance **a** directly, using **a.x = 3** and **a.y = 4**. This is made possible by the **public:** directive in the class definition. This is considered bad practice. See chapter 30.

A method is allowed to change the variables of the instance it is acting upon:

```
using namespace std;
#include <iostream>

class vector
{
public:

   double x;
   double y;

   vector its_oposite()
   {
      vector r;

      r.x = -x;
      r.y = -y;

      return r;
   }

   void be_oposited()
   {
      x = -x;
      y = -y;
   }

   void be_calculated (double a, double b, double c, double d)
   {
      x = a - c;
      y = b - d;
   }

   vector operator * (double a)
   {
      vector r;
```

```
      r.x = x * a;
      r.y = y * a;

      return r;
   }
};

int main ()
{
   vector a, b;

   a.x = 3;
   a.y = 5;

   b = a.its_oposite();

   cout << "Vector a: " << a.x << ", " << a.y << endl;
   cout << "Vector b: " << b.x << ", " << b.y << endl;

   b.be_oposited();
   cout << "Vector b: " << b.x << ", " << b.y << endl;

   a.be_calculated (7, 8, 3, 2);
   cout << "Vector a: " << a.x << ", " << a.y << endl;

   a = b * 2;
   cout << "Vector a: " << a.x << ", " << a.y << endl;

   a = b.its_oposite() * 2;
   cout << "Vector a: " << a.x << ", " << a.y << endl;

   cout << "x of oposite of a: " << a.its_oposite().x << endl;

   return 0;
}
```

Output

```
Vector a: 3, 5
Vector b: -3, -5
Vector b: 3, 5
Vector a: 4, 6
Vector a: 6, 10
Vector a: -6, -10
x of oposite of a: 6
```

18.

Very special and essential methods are the CONSTRUCTOR and DESTRUCTOR. They are automatically called whenever an instance of a class is created or destroyed (variable declaration, end of program, **new**, **delete**...).

The constructor will initialize the variables of the instance, do some calculations, allocate some memory for the instance, output some text... whatever is needed.

Here is an example of a class definition with two overloaded constructors:

```
using namespace std;
#include <iostream>

class vector
{
public:

   double x;
   double y;

   vector ()                        // same name as class
   {
      x = 0;
      y = 0;
   }

   vector (double a, double b)
   {
      x = a;
      y = b;
   }

};

int main ()
{
   vector k;                        // vector () is called

   cout << "vector k: " << k.x << ", " << k.y << endl << endl;

   vector m (45, 2);                // vector (double, double) is called

   cout << "vector m: " << m.x << ", " << m.y << endl << endl;

   k = vector (23, 2);              // vector created, copied to k, then erased

   cout << "vector k: " << k.x << ", " << k.y << endl << endl;

   return 0;
}
```

Output
vector k: 0, 0 vector m: 45, 2 vector k: 23, 2

It is good practice to try not to overload the constructors. It is best to declare only one constructor and give it default parameters wherever possible:

```
using namespace std;
#include <iostream>
```

```
class vector
{
public:

   double x;
   double y;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }
};

int main ()
{
   vector k;
   cout << "vector k: " << k.x << ", " << k.y << endl << endl;

   vector m (45, 2);
   cout << "vector m: " << m.x << ", " << m.y << endl << endl;

   vector p (3);
   cout << "vector p: " << p.x << ", " << p.y << endl << endl;

   return 0;
}
```

Output
vector k: 0, 0 vector m: 45, 2 vector p: 3, 0

The destructor is often unnecessary. You can use it to do some calculations whenever an instance is destroyed or output some text for debugging... But if variables of the instance point to some allocated memory then the role of the destructor is essential: it must free that memory! Here is an example of such an application:

```
using namespace std;
#include <iostream>
#include <cstring>

class person
{
public:

   char *name;
   int age;

   person (char *n = "no name", int a = 0)
   {
      name = new char [100];                 // better than malloc!
      strcpy (name, n);
      age = a;
      cout << "Instance initialized, 100 bytes allocated" << endl;
   }
```

```
   ~person ()                                // The destructor
   {
      delete name;                           // instead of free!

                                             // delete [] name would be more
                                             // academic but it is not vital
                                             // here since the array contains
                                             // no C++ sub-objects that need
                                             // to be deleted.

      cout << "Instance going to be deleted, 100 bytes freed" << endl;
   }
};

int main ()
{
    cout << "Hello!" << endl << endl;

    person a;
    cout << a.name << ", age " << a.age << endl << endl;

    person b ("John");
    cout << b.name << ", age " << b.age << endl << endl;

    b.age = 21;
    cout << b.name << ", age " << b.age << endl << endl;

    person c ("Miki", 45);
    cout << c.name << ", age " << c.age << endl << endl;

    cout << "Bye!" << endl << endl;

   return 0;
}
```

Output

```
Hello!

Instance initialized, 100 bytes allocated
no name, age 0

Instance initialized, 100 bytes allocated
John, age 0

John, age 21

Instance initialized, 100 bytes allocated
Miki, age 45

Bye!

Instance going to be deleted, 100 bytes freed
Instance going to be deleted, 100 bytes freed
Instance going to be deleted, 100 bytes freed
```

Here is a short example of an array class definition. A method that is an overload of the **[]** operator and outputs a reference (**&**) is used in order to generate an error if an attempt is made to access data outside the limits of an array:

```
using namespace std;
#include <iostream>
#include <cstdlib>

class array
{
public:
   int size;
   double *data;

   array (int s)
   {
      size = s;
      data = new double [s];
   }

   ~array ()
   {
      delete [] data;
   }

   double &operator [] (int i)
   {
      if (i < 0 || i >= size)
      {
         cerr << endl << "Out of bounds" << endl;
         exit (EXIT_FAILURE);
      }
      else return data [i];
   }
};

int main ()
{
   array t (5);

   t[0] = 45;                          // OK
   t[4] = t[0] + 6;                    // OK
   cout << t[4] << endl;               // OK

   t[10] = 7;                          // error!

   return 0;
}
```

Output
51 Out of bounds

19.

If you cast an object like a vector, everything will happen correctly. For example, if vector **k** contains **(4, 7)**, after the cast **m** = **k** the vector **m** will contain **(4, 7)** too. The values of **k.x** and **k.y** have simply been copied to **m.x** and **m.y**. Now suppose you're playing with objects like the **person** class above. Those objects contain a pointer to a character string. If

you cast the **person** object by writing **p = r** it is necesary that some function does the work to make **p** be a correct copy of **r**. Otherwise, **p.name** will point to the same physical character string as **r.name**. What's more, the former character string pointed to by **p.name** is lost and becomes a memory zombie. The result will be catastrophic: a mess of pointers and lost data. The methods that will do the job are the COPY CONSTRUCTOR and an overload of the = operator:

```
using namespace std;
#include <iostream>
#include <cstring>

class person
{
public:

   char *name;
   int age;

   person (char *n = "no name", int a = 0)
   {
      name = new char[100];
      strcpy (name, n);
      age = a;
   }

   person (const person &s)       // The COPY CONSTRUCTOR
   {
      name = new char[100];
      strcpy (name, s.name);
      age = s.age;
   }

   person& operator= (const person &s)  // overload of =
   {
      strcpy (name, s.name);
      age = s.age;
      return *this;
   }

   ~person ()
   {
      delete [] name;
   }
};

void modify_person (person& h)
{
   h.age += 7;
}

person compute_person (person h)
{
   h.age += 7;
   return h;
}


int main ()
{
   person p;
   cout << p.name << ", age " << p.age << endl << endl;
   // output: no name, age 0

   person k ("John", 56);
   cout << k.name << ", age " << k.age << endl << endl;
   // output: John, age 56
```

```
    p = k;
    cout << p.name << ", age " << p.age << endl << endl;
    // output: John, age 56

    p = person ("Bob", 10);
    cout << p.name << ", age " << p.age << endl << endl;
    // output: Bob, age 10

    // Neither the copy constructor nor the overload
    // of = are needed for this operation that modifies
    // p since just the reference towards p is passed to
    // the function modify_person:
    modify_person (p);
    cout << p.name << ", age " << p.age << endl << endl;
    // output: Bob, age 17

    // The copy constructor is called to pass a complete
    // copy of p to the function compute_person. The
    // function uses that copy to make its computations
    // then a copy of that modified copy is made to
    // return the result. Finaly the overload of = is
    // called to paste that second copy inside k:
    k = compute_person (p);
    cout << p.name << ", age " << p.age << endl << endl;
    // output: Bob, age 17
    cout << k.name << ", age " << k.age << endl << endl;
    // output: Bob, age 24

    return 0;
}
```

Output
no name, age 0
John, age 56
John, age 56
Bob, age 10
Bob, age 17
Bob, age 17
Bob, age 24

The copy constructor allows your program to make copies of instances when doing calculations. It is a key method. During calculations, instances are created to hold intermediate results. They are modified, cast and destroyed without you being aware. This is why those methods can be useful even for simple objects (see chapter 14.).

In all the examples above, the methods are defined inside the class definition. That automatically makes them inline methods.

20.

If a method cannot be inline, or you do not want it to be inline, or if you want the class definition to contain the minimum amount of information (or you simply want the usual separate .h header file and .cpp source code file), then you need only put the prototype of the method inside the class and define the method below the class (or in a separate .cpp source file):

```
using namespace std;
#include <iostream>

class vector
{
public:

   double x;
   double y;

   double surface();         // The ; and no {} show it is a prototype
};

double vector::surface()
{
   double s = 0;

   for (double i = 0; i < x; i++)
   {
      s = s + y;
   }

   return s;
}

int main ()
{
   vector k;

   k.x = 4;
   k.y = 5;

   cout << "Surface: " << k.surface() << endl;

   return 0;
}
```

Output
`Surface: 20`

For beginners:

If you intend to develop a serious C or C++ program, you need to separate the source code into .h header files and .cpp source files. This is a short example of how it is done. The program above is split into three files :

A header file vector.h:

```
class vector
{
public:

   double x;
   double y;

   double surface();
};
```

A source file vector.cpp:

```
using namespace std;
#include "vector.h"


double vector::surface()
{
   double s = 0;

   for (double i = 0; i < x; i++)
   {
      s = s + y;
   }

   return s;
}
```

And another source file main.cpp:

```
using namespace std;
#include <iostream>
#include "vector.h"

int main ()
{
   vector k;

   k.x = 4;
   k.y = 5;

   cout << "Surface: " << k.surface() << endl;

   return 0;
}
```

Assuming vector.cpp is perfect, you compile it once and for all into a .o "object file". The command below produces that object code file, called vector.o:

```
g++ -c vector.cpp
```

Each time you modify the main.cpp source file, you compile it into an executable file, say test20. You tell the compiler explicitly that it has to link the vector.o object file into the final test20 executable:

```
g++ main.cpp vector.o -o test20
```

Run the executable this way:

```
./test20
```

This has several advantages:

- The source code of vector.cpp need be compiled only once. This saves a lot of time on big projects. (Linking the vector.o file into the test20 executable is very fast.)
- You can give somebody the .h file and the .o file(s). That way they can use your software but not change it because they don't have the .cpp file(s) (don't rely too much on this, wait until you master these questions).

Note you can compile main.cpp too into an object file and then link it with vector.o:

```
g++ -c main.cpp

g++ main.o vector.o test20
```

This wanders away from the "differences between C and C++" topic but if you want to look like a real programmer, you need to condense the above commands into a Makefile and compile using the `make` command. The file content beneath is an oversimplified version of such a Makefile. Copy it to a file named Makefile. Please note, and this is very important, that the space before the `g++` commands is mandatory and that it is a Tab character. Do not type the spacebar here. Instead use the tabulation key (full left of your keyboard, above the caps lock).

```
test20: main.o vector.o
        g++ main.o vector.o -o test20

main.o: main.cpp vector.h
        g++ -c main.cpp

vector.o: vector.cpp vector.h
        g++ -c vector.cpp
```

In order to use that Makefile to compile, type:

```
make test20
```

The make command will parse through the file Makefile and figure out what it has to do. To start with, it's told that test20 depends on main.o and vector.o. So it will automatically launch "`make main.o`" and "`make vector.o`". Then it will check if test20 already exists and checks the date stamps of test20, main.o and vector.o. If test20 already exists and main.o and vector.o have a date stamp earlier than test20, the `make` command determines that the current version of test20 is up to date, so it has nothing to do. It will just report it did nothing. Otherwise, if test20 does not exist, or main.o or vector.o are more recent than test20, the command that creates an up-to-date version of test20 is executed: `g++ main.o vector.o -o test20`.

This next version of the Makefile is closer to a standard Makefile:

```
all: test20
```

```
test20: main.o vector.o
        g++ main.o vector.o -o test20

main.o: main.cpp vector.h
        g++ -c main.cpp

vector.o: vector.cpp vector.h
        g++ -c vector.cpp

clean:
        rm -f *.o test20 *~ #*
```

You trigger the compilation by just typing the `make` command. The first line in the Makefile implies that if you just type `make` you intended "`make test20`":

```
make
```

This command erases all the files produced during compilation and all text editor backup files:

```
make clean
```

21.

When a method is applied to an instance, that method may use the instance's variables, modify them... But sometimes it is necessary to know the address of the instance. No problem, the keyword **this** is intended for that purpose:

```
using namespace std;
#include <iostream>
#include <cmath>

class vector
{
public:

   double x;
   double y;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }

   double module()
   {
      return sqrt (x * x + y * y);
   }

   void set_length (double a = 1)
   {
      double length;

      length = this->module();
```

```
      x = x / length * a;
      y = y / length * a;
   }
};

int main ()
{
   vector c (3, 5);

   cout << "The module of vector c: " << c.module() << endl;

   c.set_length(2);            // Transforms c in a vector of size 2.

   cout << "The module of vector c: " << c.module() << endl;

   c.set_length();             // Transforms b in an unitary vector.

   cout << "The module of vector c: " << c.module() << endl;

   return 0;
}
```

Output
The module of vector c: 5.83095 The module of vector c: 2 The module of vector c: 1

22.

Of course, it is possible to declare arrays of objects:

```
using namespace std;
#include <iostream>
#include <cmath>

class vector
{
public:

   double x;
   double y;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }

   double module ()
   {
      return sqrt (x * x + y * y);
   }
```

```
};

int main ()
{
   vector s [1000];

   vector t[3] = {vector(4, 5), vector(5, 5), vector(2, 4)};

   s[23] = t[2];

   cout << t[0].module() << endl;

   return 0;
}
```

Output
6.40312

23.

Here is an example of a full class declaration:

```
using namespace std;
#include <iostream>
#include <cmath>

class vector
{
public:

   double x;
   double y;

   vector (double = 0, double = 0);

   vector operator + (vector);
   vector operator - (vector);
   vector operator - ();
   vector operator * (double a);
   double module();
   void set_length (double = 1);
};

vector::vector (double a, double b)
{
   x = a;
   y = b;
}

vector vector::operator + (vector a)
{
   return vector (x + a.x, y + a.y);
}
```

```
vector vector::operator - (vector a)
{
   return vector (x - a.x, y - a.y);
}

vector vector::operator - ()
{
   return vector (-x, -y);
}

vector vector::operator * (double a)
{
   return vector (x * a, y * a);
}

double vector::module()
{
   return sqrt (x * x + y * y);
}

void vector::set_length (double a)
{
   double length = this->module();

   x = x / length * a;
   y = y / length * a;
}

ostream& operator << (ostream& o, vector a)
{
   o << "(" << a.x << ", " << a.y << ")";
   return o;
}

int main ()
{
   vector a;
   vector b;
   vector c (3, 5);

   a = c * 3;
   a = b + c;
   c = b - c + a + (b - a) * 7;
   c = -c;

   cout << "The module of vector c: " << c.module() << endl;

   cout << "The content of vector a: " << a << endl;
   cout << "The oposite of vector a: " << -a << endl;

   c.set_length(2);            // Transforms c in a vector of size 2.

   a = vector (56, -3);
   b = vector (7, c.y);

   b.set_length();             // Transforms b in an unitary vector.

   cout << "The content of vector b: " << b << endl;

   double k;
   k = vector(1, 1).module();  // k will contain 1.4142.
   cout << "k contains: " << k << endl;

   return 0;
}
```

Output

```
The module of vector c: 40.8167
The content of vector a: (3, 5)
The oposite of vector a: (-3, -5)
The content of vector b: (0.971275, 0.23796)
k contains: 1.41421
```

It is also possible to define a function to produces the sum of two vectors without mentioning it inside the vector class definition. Then it will not be a method of the class vector, but rather just a function that uses vectors:

```
vector operator + (vector a, vector b)
{
   return vector (a.x + b.x, a.y + b.y);
}
```

In the example of a full class definition, above, the multiplication of a vector by a double is defined. Suppose we want the multiplication of a double by a vector to be defined too. Then we must write an isolated function outside the class:

```
vector operator * (double a, vector b)
{
   return vector (a * b.x, a * b.y);
}
```

Of course the keywords **new** and **delete** work for class instances too. What's more, **new** automatically calls the constructor in order to initialize the objects, and **delete** automatically calls the destructor before deallocating the memory the instance variables take:

```
using namespace std;
#include <iostream>
#include <cmath>

class vector
{
public:

   double x;
   double y;

   vector (double = 0, double = 0);

   vector operator + (vector);
   vector operator - (vector);
   vector operator - ();
   vector operator * (double);
   double module();
   void set_length (double = 1);
};
```

```
vector::vector (double a, double b)
{
   x = a;
   y = b;
}

vector vector::operator + (vector a)
{
   return vector (x + a.x, y + a.y);
}

vector vector::operator - (vector a)
{
   return vector (x - a.x, y - a.y);
}

vector vector::operator - ()
{
    return vector (-x, -y);

}

vector vector::operator * (double a)
{
    return vector (a * x, a * y);
}

double vector::module()
{
   return sqrt (x * x + y * y);
}

void vector::set_length (double a)
{
   vector &the_vector = *this;

   double length = the_vector.module();

   x = x / length * a;
   y = y / length * a;
}

ostream& operator << (ostream& o, vector a)
{
   o << "(" << a.x << ", " << a.y << ")";
   return o;
}

int main ()
{
   vector c (3, 5);

   vector *r;                  // r is a pointer to a vector.

   r = new vector;             // new allocates the memory necessary
   cout << *r << endl;         // to hold a vectors' variable,
                               // calls the constructor who will
                               // initialize it to 0, 0. Then finally
                               // new returns the address of the vector.

   r->x = 94;
   r->y = 345;
   cout << *r << endl;

   *r = vector (94, 343);
   cout << *r << endl;

   *r = *r - c;
```

```
    r->set_length(3);
    cout << *r << endl;

    *r = (-c * 3  +  -*r * 4) * 5;
    cout << *r << endl;

    delete r;                   // Calls the vector destructor then
                                // frees the memory.

    r = &c;                     // r points towards vector c
    cout << *r << endl;

    r = new vector (78, 345);   // Creates a new vector.
    cout << *r << endl;         // The constructor will initialise
                                // the vector's x and y at 78 and 345

    cout << "x component of r: " << r->x << endl;
    cout << "x component of r: " << (*r).x << endl;

    delete r;

    r = new vector[4];          // creates an array of 4 vectors

    r[3] = vector (4, 5);
    cout << r[3].module() << endl;

    delete [] r;                // deletes the array

    int n = 5;
    r = new vector[n];          // Cute!

    r[1] = vector (432, 3);
    cout << r[1] << endl;

    delete [] r;

    return 0;
}
```

Output

```
(0, 0)
(94, 345)
(94, 343)
(0.77992, 2.89685)
(-60.5984, -132.937)
(3, 5)
(78, 345)
x component of r: 78
x component of r: 78
6.40312
(432, 3)
```

24.

One or more variables in a class can be declared **static**. In which case, only one instance of those variables exist, shared

by all instances of the class. It must be initialised outside the class declaration :

```
using namespace std;
#include <iostream>

class vector
{
public:

   double x;
   double y;
   static int count;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
      count++;
   }

   ~vector()
   {
      count--;
   }
};

int vector::count = 0;

int main ()
{
   cout << "Number of vectors:" << endl;

   vector a;
   cout << vector::count << endl;

   vector b;
   cout << vector::count  << endl;

   vector *r, *u;

   r = new vector;
   cout << vector::count << endl;

   u = new vector;
   cout << a.count << endl;

   delete r;
   cout << vector::count << endl;

   delete u;
   cout << b.count << endl;

   return 0;
}
```

Output
1 2 3 4 3

25.

A class variable can also be **const**ant. That's just like static, except it is given a value inside the class declaration and that value cannot be modified:

```
using namespace std;
#include <iostream>

class vector
{
public:

   double x;
   double y;
   const static double pi = 3.1415927;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }

   double cilinder_volume ()
   {
      return x * x / 4 * pi * y;
   }
};

int main()
{
   cout << "The value of pi: " << vector::pi << endl << endl;

   vector k (3, 4);

   cout << "Result: " << k.cilinder_volume() << endl;

   return 0;
}
```

Output
The value of pi: 3.14159 Result: 28.2743

26.

A class can be DERIVED from another class. The new class INHERITS the variables and methods of the BASE CLASS. Additional variables and/or methods can be added:

```
using namespace std;
#include <iostream>
#include <cmath>

class vector
{
public:

   double x;
   double y;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }

   double module()
   {
      return sqrt (x*x + y*y);
   }

   double surface()
   {
       return x * y;
   }
};

class trivector: public vector    // trivector is derived from vector
{
public:
   double z;                      // added to x and y from vector

   trivector (double m=0, double n=0, double p=0): vector (m, n)
   {
      z = p;                      // Vector constructor will
   }                              // be called before trivector
                                  // constructor, with parameters
                                  // m and n.

   trivector (vector a)           // What to do if a vector is
   {                              // cast to a trivector
      x = a.x;
      y = a.y;
      z = 0;
   }

   double module ()               // define module() for trivector
   {
      return sqrt (x*x + y*y + z*z);
   }

   double volume ()
   {
       return this->surface() * z;         // or x * y * z
   }
};
```

```
int main ()
{
   vector a (4, 5);
   trivector b (1, 2, 3);

   cout << "a (4, 5)    b (1, 2, 3)    *r = b" << endl << endl;

   cout << "Surface of a: " << a.surface() << endl;
   cout << "Volume of b: " << b.volume() << endl;
   cout << "Surface of base of b: " << b.surface() << endl;

   cout << "Module of a: " << a.module() << endl;
   cout << "Module of b: " << b.module() << endl;
   cout << "Module of base of b: " << b.vector::module() << endl;

   trivector k;
   k = a;               // thanks to trivector(vector) definition
                        // copy of x and y,       k.z = 0
   vector j;
   j = b;               // copy of x and y.       b.z leaved out

   vector *r;
   r = &b;

   cout << "Surface of r: " << r->surface() << endl;
   cout << "Module of r: " << r->module() << endl;

   return 0;
}
```

Output

```
a (4, 5)    b (1, 2, 3)    *r = b

Surface of a: 20
Volume of b: 6
Surface of base of b: 2
Module of a: 6.40312
Module of b: 3.74166
Module of base of b: 2.23607
Surface of r: 2
Module of r: 2.23607
```

27.

In the program above, **r->module()** calculates the vector module, using **x** and **y**, because **r** has been declared a vector pointer. The fact that **r** actually points to a trivector is not taken into account. If you want the program to check the type of the pointed object and choose the appropriate method, then you must declare that method as **virtual** inside the base class.

(If at least one of the methods of the base class is virtual then a "header" of 4 bytes is added to every instance of the classes. This allows the program to determine what a vector actually points to.) (4 bytes is probably implementation specific. On a 64 bit machine maybe it is 8 bytes...)

```
using namespace std;
#include <iostream>
#include <cmath>

class vector
{
public:

   double x;
   double y;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }

   virtual double module()
   {
      return sqrt (x*x + y*y);
   }
};

class trivector: public vector
{
public:
   double z;

   trivector (double m = 0, double n = 0, double p = 0)
   {
      x = m;                  // Just for the game,
      y = n;                  // here I do not call the vector
      z = p;                  // constructor and I make the
   }                          // trivector constructor do the
                              // whole job. Same result.

   double module ()
   {
      return sqrt (x*x + y*y + z*z);
   }
};

void test (vector &k)
{
    cout << "Test result:          " << k.module() << endl;
}

int main ()
{
   vector a (4, 5);
   trivector b (1, 2, 3);

   cout << "a (4, 5)    b (1, 2, 3)" << endl << endl;

   vector *r;

   r = &a;
   cout << "module of vector a: " << r->module() << endl;

   r = &b;
   cout << "module of trivector b: " << r->module() << endl;

   test (a);

   test (b);

   vector &s = b;
```

```
   cout << "module of trivector b: " << s.module() << endl;

   return 0;
}
```

Output
a (4, 5) b (1, 2, 3) module of vector a: 6.40312 module of trivector b: 3.74166 Test result: 6.40312 Test result: 3.74166 module of trivector b: 3.74166

28.

Maybe you wonder if a class can be derived from more than one base class. The answer is yes:

```
using namespace std;
#include <iostream>
#include <cmath>

class vector
{
public:

   double x;
   double y;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }

   double surface()
   {
      return fabs (x * y);
   }
};

class number
{
public:

   double z;

   number (double a)
   {
      z = a;
   }

   int is_negative ()
```

```
   {
      if (z < 0) return 1;
      else       return 0;
   }
};

class trivector: public vector, public number
{
public:

   trivector(double a=0, double b=0, double c=0): vector(a,b), number(c)
   {
   }           // The trivector constructor calls the vector
               // constructor, then the number constructor,
               // and in this example does nothing more.

   double volume()
   {
      return fabs (x * y * z);
   }
};

int main ()
{
   trivector a(2, 3, -4);

   cout << a.volume() << endl;
   cout << a.surface() << endl;
   cout << a.is_negative() << endl;

   return 0;
}
```

Output
24 6 1

29.

Class derivation allows you to construct more complex classes built from base classes. There is another application of class derivation: allowing the programmer to write generic functions.

Suppose you define a base class with no variables. It makes no sense to use instances of that class inside your program. But then you write a function whose purpose it is to sort instances of that class. That function will be able to sort any type of object provided it belongs to a class derived from that base class! The only condition is that inside of each derived class definition, all methods that the sort function needs are correctly defined:

```
using namespace std;
#include <iostream>
#include <cmath>
```

```
class octopus
{
public:

   virtual double module() = 0;  // = 0 implies function is not
                                 // defined. This makes instances
                                 // of this class cannot be declared.
};

double biggest_module (octopus &a, octopus &b, octopus &c)
{
    double r = a.module();
    if (b.module() > r) r = b.module();
    if (c.module() > r) r = c.module();
    return r;
}

class vector: public octopus
{
public:

   double x;
   double y;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }

   double module()
   {
      return sqrt (x * x + y * y);
   }
};

class number: public octopus
{
public:

   double n;

   number (double a = 0)
   {
      n = a;
   }

   double module()
   {
      if (n >= 0) return n;
      else        return -n;
   }
};

int main ()
{
    vector k (1,2), m (6,7), n (100, 0);
    number p (5),   q (-3),  r (-150);

    cout << biggest_module (k, m, n) << endl;
    cout << biggest_module (p, q, r) << endl;

    cout << biggest_module (p, q, n) << endl;

   return 0;
}
```

Output
100 150 100

Perhaps you think "okay, that's a good idea to derive classes from the class **octopus** because that way I can apply it to instances of my class's methods and function that were designed in a generic way for the **octopus** class. But what if there's another base class, named **cuttlefish**, which has very interesting methods and functions too? Do I have to make my choice between **octopus** and **cuttlefish** when I want to derive a class?" No, of course not. A derived class can be derived from both **octopus** and **cuttlefish**. That's POLYMORPHISM. The derived class simply has to define the methods necessary for **octopus** together with the methods necessary for **cuttlefish**:

```
class octopus
{
   virtual double module() = 0;
};

class cuttlefish
{
   virtual int test() = 0;
};

class vector: public octopus, public cuttlefish
{
   double x;
   double y;

   double module ()
   {
      return sqrt (x * x + y * y);
   }

   int test ()
   {
      if (x > y) return 1;
      else       return 0;
   }
}
```

30.

The **public:** directive means the variables or the methods below can be accessed and used everywhere in the program.

If you want the variables and methods to be accessible only to methods of the class AND to methods of derived classes, then you must put the keyword **protected:** before them.

If you want variables or methods to be accessible ONLY to methods of the class, then you must put the keyword **private:** before them.

The fact that variables or methods are declared private or protected means that nothing external to the class can access or use them. That's ENCAPSULATION. (If you want to give a specific function the right to access those variables and methods, then you must include that function's prototype inside the class definition, preceded by the keyword **friend**.)

Good practice is to encapsulate all the variables of a class. This can sound strange if you're used to structs in C. Indeed a struct only makes sense if you can access its data... In C++ you have to create methods to acces the data inside a class. The example below uses the basic example of chapter 17, yet declares the class data to be protected:

```
using namespace std;
#include <iostream>

class vector
{
protected:

   double x;
   double y;

public:

   void set_x (int n)
   {
      x = n;
   }

   void set_y (int n)
   {
      y = n;
   }

   double surface ()
   {
      double s;
      s = x * y;
      if (s < 0) s = -s;
      return s;
   }
};

int main ()
{
   vector a;

   a.set_x (3);
   a.set_y (4);

   cout << "The surface of a: " << a.surface() << endl;

   return 0;
}
```

Output
The surface of a: 12

The example above is a bit odd since the class data x and y can be set but they cannot be read back. Any attempt in function main () to read **a.x** or **a.y** will result in a compilation error. In the next example, x and y can be read back:

```
using namespace std;
#include <iostream>

class vector
{
protected:

   double x;
   double y;

public:

   void set_x (int n)
   {
      x = n;
   }

   void set_y (int n)
   {
      y = n;
   }

   double get_x ()
   {
      return x;
   }

   double get_y ()
   {
      return y;
   }

   double surface ()
   {
      double s;
      s = x * y;
      if (s < 0) s = -s;
      return s;
   }
};

int main ()
{
   vector a;

   a.set_x (3);
   a.set_y (4);

   cout << "The surface of a: " << a.surface() << endl;
   cout << "The width of a:   " << a.get_x() << endl;
   cout << "The height of a:  " << a.get_y() << endl;

   return 0;
}
```

Output

```
The surface of a: 12
The width of a:   3
The height of a:  4
```

In C++ one is not supposed to access the data of a class directly. Methods have to be declared. Why is this? Many reasons exist. One is that this allows you to change the way the data is represented within the class. Another reason is this allows data inside the class to be cross-dependent. Suppose **x** and **y** must always be of the same sign, otherwise ugly things can happen... If one is allowed to access the class data directly, it would be easy to impose say a positive x and a negative y. In the example below, this is strictly controlled:

```
using namespace std;
#include <iostream>

int sign (double n)
{
   if (n >= 0) return 1;
   return -1;
}

class vector
{
protected:

   double x;
   double y;

public:

   void set_x (int n)
   {
      x = n;
      if (sign (x) != sign(y)) y = -y;
   }

   void set_y (int n)
   {
      y = n;
      if (sign (y) != sign(x)) x = -x;
   }

   double get_x ()
   {
      return x;
   }

   double get_y ()
   {
      return y;
   }

   double surface ()
   {
      double s;
      s = x * y;
      if (s < 0) s = -s;
      return s;
   }
};

int main ()
{
   vector a;

   a.set_x (-3);
   a.set_y (4);

   cout << "The surface of a: " << a.surface() << endl;
   cout << "The width of a:   " << a.get_x() << endl;
   cout << "The height of a:  " << a.get_y() << endl;
```

```
   return 0;
}
```

Output
The surface of a: 12 The width of a: 3 The height of a: 4

31.

Let's talk about input/output. In C++ that's a very broad subject.

Here is a program that writes to a file:

```
using namespace std;
#include <iostream>
#include <fstream>

int main ()
{
   fstream f;

   f.open("test.txt", ios::out);

   f << "This is a text output to a file." << endl;

   double a = 345;

   f  << "A number: " << a << endl;

   f.close();

   return 0;
}
```

Content of file test.txt
This is a text output to a file. A number: 345

Here is a program that reads from a file:

```
using namespace std;
```

```
#include <iostream>
#include <fstream>

int main ()
{
   fstream f;
   char c;

   cout << "What's inside the test.txt file" << endl;
   cout << endl;

   f.open("test.txt", ios::in);

   while (! f.eof() )
   {
      f.get(c);                          // Or c = f.get()
      cout << c;
   }

   f.close();

   return 0;
}
```

Output
This is a text output to a file. A number: 345

32.

Generally speaking, it is possible to do on character arrays the same operations as on files. This is very useful to convert data or manage memory arrays.

Here is a program that writes inside a character array:

```
using namespace std;
#include <iostream>
#include <strstream>
#include <cstring>
#include <cmath>

int main ()
{
   char a[1024];
   ostrstream b(a, 1024);

   b.seekp(0);                           // Start from first char.
   b << "2 + 2 = " << 2 + 2 << ends;     // ( ends, not endl )
                                         // ends is simply the
                                         // null character   '\0'
   cout << a << endl;

   double v = 2;
```

```
   strcpy (a, "A sinus: ");

   b.seekp(strlen (a));
   b << "sin (" << v << ") = " << sin(v) << ends;

   cout << a << endl;

   return 0;
}
```

Output
2 + 2 = 4 A sinus: sin (2) = 0.909297

A program that reads from a character string:

```
using namespace std;
#include <iostream>
#include <strstream>
#include <cstring>

int main ()
{
   char a[1024];
   istrstream b(a, 1024);

   strcpy (a, "45.656");

   double k, p;

   b.seekg(0);                          // Start from first character.
   b >> k;

   k = k + 1;

   cout << k << endl;

   strcpy (a, "444.23 56.89");

   b.seekg(0);
   b >> k >> p;

   cout << k << ", " << p + 1 << endl;

   return 0;
}
```

Output
46.656 444.23, 57.89

33.

This program performs formatted output two different ways. Please note the **width()** and **setw()** MODIFIERS are only effective on the next item output to the stream. Subsequent items will not be influenced.

```
using namespace std;
#include <iostream>
#include <iomanip>

int main ()
{
   int i;

   cout << "A list of numbers:" << endl;
   for (i = 1; i <= 1024; i *= 2)
   {
      cout.width (7);
      cout << i << endl;
   }

   cout << "A table of numbers:" << endl;
   for (i = 0; i <= 4; i++)
   {
      cout << setw(3) << i << setw(5) << i * i * i << endl;
   }

   return 0;
}
```

Output

```
A list of numbers:
      1
      2
      4
      8
     16
     32
     64
    128
    256
    512
   1024
A table of numbers:
  0    0
  1    1
  2    8
  3   27
  4   64
```

You now have a basic knowledge about C++. Inside good books you will learn many more things. The file management system is very powerful, it has many other possibilities than those illustrated here. There is also a lot more to say about classes: template classes, virtual classes...

In order to work efficiently with C++ you will need a good reference book, just like you need one for C. You will also need information on how C++ is used in your particular domain of activity. The standards, the global approach, the tricks, the typical problems encountered and their solutions... The best reference is of course the books written by Bjarne Stroustrup himself (I don't recall which of them I read). The following book contains almost every detail about C and C++ and is constructed in a way similar to this text:

Jamsa's C/C++ Programmer's Bible
&copyright; 1998 Jamsa Press
Las Vegas, United States

French edition:
C/C++ La Bible du programmeur
Kris Jamsa, Ph.D - Lars Klander
France : Editions Eyrolles
www.eyrolles.com
Canada : Les Editions Reynald Goulet inc.
www.goulet.ca
ISBN 2-212-09058-7

Other reference:

accu: www.accu.org/bookreviews/public/reviews/0hr/index.htm

CoderSource.net: www.codersource.net/

C++ Guide: google-styleguide.googlecode.com/svn/trunk/cppguide.xml

A similar tutorial for Ada is available at www.adahome.com/Ammo/cpp2ada.html

A Haskell tutorial by a C programmer: learnyouahaskell.com

This page **translated in Belorussian**: www.webhostinghub.com/support/cppcen-be

I wish to thank Didier Bizzarri, Toni Ronkko, Frédéric Cloth, Jack Lam, Morten Brix Pedersen, Elmer Fittery, Ana Yuseepi, William L. Dye, Bahjat F. Qaqish, Muthukumar Veluswamy, Marco Cimarosti, Jarrod Miller, Nikolaos Pothitos, Ralph Wu, Dave Abercrombie, Alex Pennington, Scott Marsden, Robert Krten, Dave Panter, Cihat Imamoglu and Bohdan Zograf for their inspiration, advice, help, data, bug reports, references, enhancement of the English wording and translation.

Eric Brasseur - February 23 1998 till June 24 2010